

# Um controle de versões refinado e flexível para artefatos de software

DANIEL CÁRNIO JUNQUEIRA  
RENATA PONTIN DE MATTOS FORTES

Laboratório Intermídia  
Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo  
Av. do Trabalhador São-Carlense, 400 - Centro - Cx. Postal 668  
São Carlos - São Paulo - Brasil CEP 13560-970  
{danielcj,renata}@icmc.usp.br

**Resumo.** Controle de versões é uma atividade essencial para a produção de software com qualidade. A estrutura utilizada pelos sistemas de controle de versões é a mesma que é utilizada pelos sistemas de arquivos, mas muitas vezes a abstração realizada para desenvolvimento de software leva em consideração o conteúdo dos arquivos e sua estrutura, incluindo detalhes como classes, métodos, blocos de controle entre outros. Ferramentas de controle de versões refinado são capazes de proporcionar um controle de versões mais detalhado. As ferramentas e modelos tradicionais, entretanto, fornecem baixa flexibilidade e apresentam alto custo e impacto de implantação em ambientes de desenvolvimento de software. Neste artigo, são apresentados um modelo e uma ferramenta que visam a oferecer suporte para realização de atividades de controle de versões refinado e flexível

**Palavras-chave.** controle de versões, controle de versões refinado, gerenciamento de configuração de software.

**Abstract.** Version control is an activity very important for high-quality software production. The structure used by version control systems is the same used by file systems, but in general the abstraction level made by software developers considers the file contents and its internal structure, including details as classes, methods, control blocks and others. Fine-grained version control tools can provide a more detailed version control. However traditional tools and models provide very low flexibility and present high cost and impact of deployment in software development environments. In this paper, there are presented a model and a tool which aim at providing support to fine-grained version control activities.

**Keywords.** version control, fine-grained version control, software configuration management.

## 1 Introdução

Controle de versões, no contexto de desenvolvimento de artefatos de software, refere-se à denominação para as técnicas e ferramentas<sup>1</sup> utilizadas para controlar a evolução de arquivos de computador. Os

registros das primeiras atividades de controle de versões datam por volta de 1950, época em que eram realizadas de forma manual [Berlack, 1991]. Em 1972 foi lançado o sistema SCCS, que é considerada a primeira ferramenta de controle de versões [Glasser, 1978].

<sup>1</sup>Neste artigo, os termos “ferramentas de controle de versões” e “sistemas de controle de versões” serão utilizados de forma indistinta para denotar o conjunto de programas de computador que automatiza as tarefas para controle de versões

Apesar das evoluções tecnológicas e das frequentes alterações nos ambientes de negócio em que as técnicas de controle de versões são utilizadas [Estublier et al., 2005], a forma como as versões são controladas em sistemas de controle de versões mantiveram-se constantes desde a primeira ferramenta [Conradi & Westfechtel, 1998]. Novos sistemas foram criados, mas, em geral, foram melhoradas as técnicas de armazenamento, controle de acesso, distribuição de repositórios, e integração com outras ferramentas para realizar os processos integrantes de GCS – mas a forma como as versões de código fonte são controladas permanece quase inalterada há três décadas, e pouca ou nenhuma inteligência foi adicionada aos sistemas de controle de versões e mesmo aos processos de GCS [Keyes, 2004].

Pesquisadores têm investido em novas estratégias de controle de versões com o objetivo fornecer melhor controle sobre os documentos ou arquivos de código fonte que estão no repositório de controle de versões, permitindo um mapeamento adequado da estrutura dos arquivos de computador no sistema de controle de versões [Lin & Reiss, 1996, Nguyen et al., 2004b] e, conseqüentemente, melhores informações para que se realize o rastreamento mais detalhado da evolução dos softwares. Entretanto, os sistemas propostos têm limitações relacionadas à flexibilidade que oferecem para configuração de como é feito o rastreamento da estrutura dos arquivos e também têm grande dependência de softwares específicos para interação com o usuário, gerando alto impacto no caso de implantação.

Neste trabalho são apresentados um modelo para realização de controle de versões refinado de forma flexível e independente de interface gráfica, além de um sistema que valida este modelo, chamado *Phoca*. A partir do estudo dos principais modelos propostos para a realização de atividades de controle de versões refinado, além do levantamento das principais características dos sistemas de controle de versão com repositórios centralizados e distribuídos, foram projetados o modelo e o sistema *Phoca*, cujo principal requisito não-funcional é a flexibilidade.

O restante deste artigo está organizado da seguinte forma: na Seção 2 é apresentada uma revisão sobre os principais conceitos e sistemas de controle de versões, além das considerações sobre a relação existente entre as atividades de controle de versões e o processo de GCS; na Seção 3 é apresentada uma revisão da literatura sobre os principais sistemas de controle de versões refinado; na Seção 4 é descrito o modelo proposto assim como o sistema *Phoca*, que valida o modelo; por fim, na Seção 5 são apresentadas as principais conclusões e são indicados possíveis

trabalhos futuros.

## 2 Controle de versões

Controle de versões refere-se à técnica de controlar a evolução de conteúdo criado ao longo do tempo, permitindo a recuperação de dados históricos, diferenças entre versões, e detalhes sobre a evolução de determinado conteúdo que tenha tido suas versões controladas [Tichy, 1982]. O fluxo das atividades de controle de versões em sistemas atuais envolve a criação de um repositório, importação de arquivos de determinado projeto, e, então, realização das atividades do ciclo *checkout-merge-commit* [IEEE Std 610, 1991, Tichy, 1982, Ambriola et al., 1990].

Para correto entendimento das atividades envolvidas, é importante observar como estão organizados os arquivos quando é utilizado um sistema de controle de versões. São utilizados repositórios de arquivos que contêm todos os arquivos de determinado projeto que estiverem sob controle de versões, além de todas as versões destes arquivos. Entretanto, nenhuma atividade é realizada diretamente sobre os arquivos do repositório – ao invés disso, cada usuário possui uma cópia de determinada versão dos arquivos, geralmente a última. É justamente para se obter esta cópia local que é utilizado o comando **checkout**.

Como vários usuários podem efetuar alterações num mesmo arquivo de forma concorrente, é importante e necessário realizar a atividade de **merge** ou intercalação das mudanças antes que cada usuário possa submeter suas alterações para o repositório. Para isso, é utilizado o comando **update** ou similar.

Durante a operação de *merge*, pode ocorrer um conflito. Na Figura 2 é ilustrado um exemplo de merge bem sucedido e, na Figura 2 é ilustrado um conflito. É possível observar que o conflito ocorre nos casos em que dois usuários editam uma mesma versão de um arquivo, ao mesmo tempo, e em regiões idênticas ou próximas, o que impossibilita a realização de intercalação automática pelo sistema de controle de versões.

Por fim, após realizadas todas as alterações e as atividades de intercalação com eventuais versões produzidas por outros usuários – que pode ocorrer de forma automática ou conflitante, exigindo intervenção manual –, um usuário pode enviar suas alterações, utilizando a operação conhecida como **commit**.

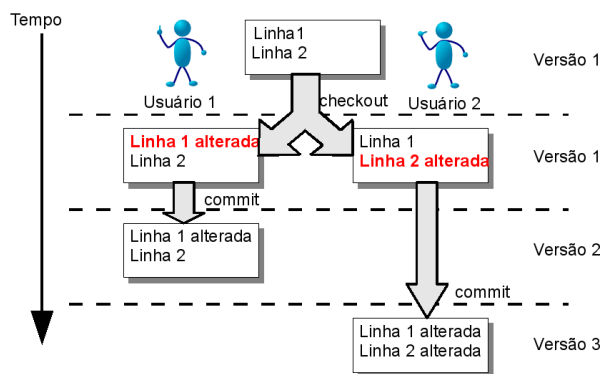


FIGURA 1: Exemplo de *merge* bem sucedido

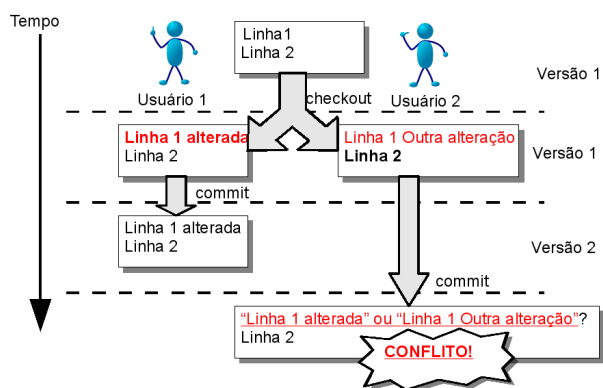


FIGURA 2: Exemplo de situação de conflito - não é possível realizar a operação de *merge*

Na Figura 2 é exibida a organização de um sistema de controle de versões com repositório centralizado. Além deste tipo de organização, os sistemas também podem ter repositórios distribuídos - sendo que, neste caso, cada usuário realiza as operações em repositórios próprios, realizando atividades de sincronização entre repositórios quando desejado.

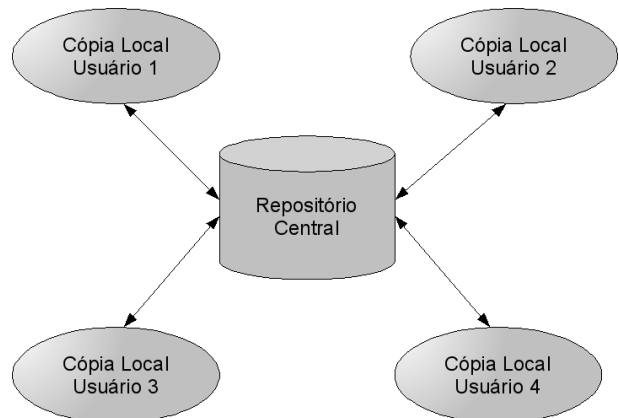


FIGURA 3: Exemplo de configuração típica do repositório para sistema de controle de revisão centralizado

Além da organização do trabalho, rastreamento e concorrência, que são permitidos pelos sistemas de controle de versões, também existe uma melhoria em relação à otimização de armazenamento e possibilidades de organização das versões. Isso ocorre porque os sistemas de controle de versões utilizam árvores de versões - o que possibilita organizar melhor as versões dos arquivos num formato de árvore, incluindo versões que evoluem de forma paralela - , além da utilização do conceito de *deltas* - diferenças entre versões, otimizando o espaço de armazenamento.

## 2.1 Ferramentas de Controle de Versões

As ferramentas de controle de versões evoluíram cronologicamente da forma mostrada na Figura 2.1.

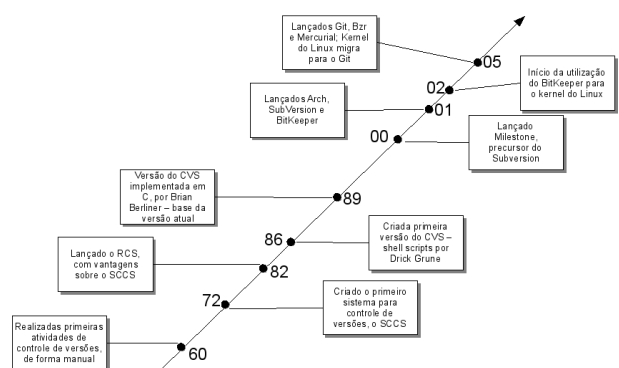


FIGURA 4: Linha do tempo das ferramentas de controle de versões

A primeira ferramenta da qual se tem conhecimento é a SCCS, criada em 1972 por Marc Rochkind

[Rochkind, 1975]. Esta ferramenta era utilizada para desenvolvimento de software e controle de versões de outros arquivos de computador que eram artefatos de projetos, e os arquivos eram todos armazenados em discos compartilhados de *mainframes*. Esta ferramenta permitia a realização de operações básicas sobre os arquivos, e não existia ainda o conceito de repositórios – cada arquivo era armazenado num diretório, e o arquivo de controle de versões correspondente era armazenado no mesmo repositório. Além disso, SCCS utilizava o conceito de “delta positivos”, segundo o qual a primeira versão de cada arquivo é armazenada de forma integral e são armazenadas as diferenças para cada versão posterior – técnica que torna mais rápida a recuperação de versões mais antigas. Na Figura 2.1 é ilustrada uma árvore de revisões criada pelo SCCS.

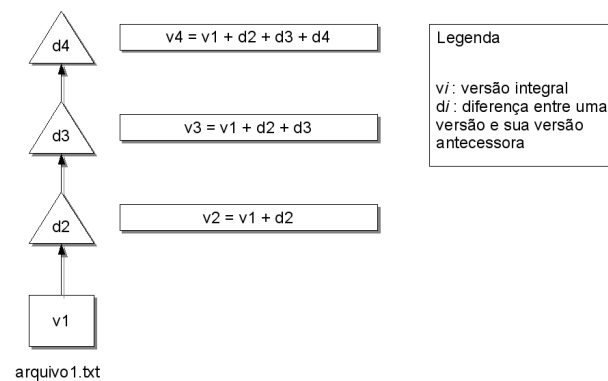


FIGURA 5: Exemplo de árvore de revisões utilizada pelo SCCS

Dez anos após o lançamento do SCCS, foi lançada a primeira versão do RCS, em 1982, de autoria de Walter Tichy [Tichy, 1982]. O projeto do RCS foi inspirado no SCCS, incorporando algumas melhorias sobre ele, incluindo uma interface com o usuário facilitada e melhoria no mecanismo de armazenamento das versões para recuperação mais fácil e rápida de informações [RCS, 2007]. Além disso, o projeto foi lançado sob uma licença de código aberto – atualmente é distribuído sobre a versão 2 da GPL. Existem versões para Unix, Windows e Mac OS X [RCS, 2007]. O RCS introduziu o conceito de “delta negativo” ou “delta reverso”. Este mecanismo consiste em manter no arquivo de controle do RCS a última versão existente – conhecida por HEAD – de forma integral, e armazenar as diferenças para as versões anteriores no tronco principal. Os deltas positivos são então utilizados para se obter as versões das ramificações. Na Figura 2.1 é exibido um exemplo de árvore de revisões armazenada para determinado arquivo. Nela é possível observar que a

última versão do tronco principal é armazenada integralmente; para se calcular uma versão anterior, por exemplo a 1.3, basta subtrair a diferença entre a 1.3 e a 1.4. Utilizando esta abordagem, as versões mais antigas exigem mais processamento, o que, geralmente, é uma situação conveniente, pois a maioria das operações concentra-se em versões recentes.

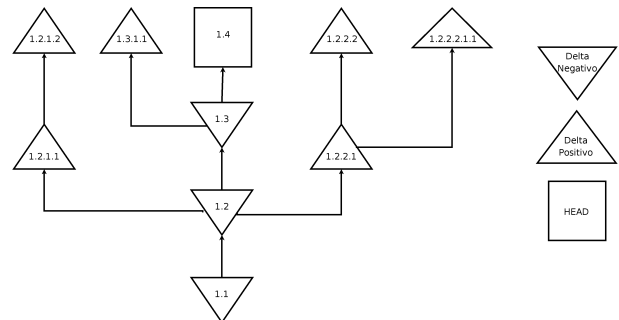


FIGURA 6: Árvore de revisões em cada item armazenado no repositório, utilizando deltas (adaptado de [Tichy, 1982])

Quatro anos após a criação do RCS, surgiu o CVS [Cederqvist et al., 2007], com o conceito de repositórios. Nesta época, evoluções ocorriam nos ambientes de desenvolvimento e via-se a popularização de computadores pessoais. Por este motivo, foi importante o conceito de repositórios – com CVS, cada desenvolvedor tem sua área de trabalho, e o controle é realizado no repositório. O CVS estabeleceu-se como o sistema padrão de controle de versões por quase duas décadas [Reis, 2003].

As principais inovações em sistemas de controle de versões criadas vieram em 2002, com os sistemas de controle de versões distribuídos. Em 2002, foi lançado, por exemplo, o Bitkeeper [Kroah-Hartman, 2002], que passou a ser utilizado para o desenvolvimento do kernel do Linux. No entanto, em abril de 2005 a permissão concedida pela BitMover de utilização gratuita do sistema para o kernel do Linux e projetos de código aberto foi suspensa, e foram então lançados três sistemas para controle de versões, todos inspirados nas idéias do Arch e do BitKeeper: o Bazaar [Blackwell et al., 2006], que é uma reimplementação do Arch, o Git, projeto liderado por Linus Torvalds na época e que hoje é utilizado para controle de versões do kernel do Linux [Garzik, 2005], e, por fim, o Mercurial, bastante influenciado pelo BitKeeper [Mercurial, 2007]. Nestes sistemas, cada desenvolvedor pode submeter suas alterações em repositórios próprios, e são oferecidas operações para sincronização de repositórios.

Atualmente, estes são os sistemas de controle de versões mais novos que existem. Suas funcionalidades

dades são semelhantes às dos sistemas BitKeeper e Arch, mas todos eles são projetos de código aberto, diferentemente do BitKeeper. Paralelamente a estes sistemas, foram apresentadas propostas de sistemas de controle de versões refinado, cujos conceitos e principais ferramentas serão apresentados na próxima seção.

### 3 Controle de versões refinado

Controle de versões é uma atividade de suporte importante para Gerência de Configuração de Software. A maioria dos sistemas de controle de versões mantém um registro das mudanças (“deltas”) entre as diferentes versões dos arquivos sob controle de versões, o que provê a base para rastrear a evolução de um sistema durante seu ciclo de vida. Entretanto, os sistemas de controle de versões geralmente realizam o controle de versões sobre os arquivos num nível de detalhes muitas vezes inadequado, geralmente em documentos inteiros ou módulos inteiros [Lindsay et al., 1997]. Mesmo os sistemas de controle de versões mais populares atualmente, como SubVersion e CVS – de repositório centralizado – e o Bazaar e o Mercurial – de repositório distribuído – ainda seguem este paradigma: o refinamento de versões em níveis de arquivos e módulos.

Pesquisadores têm investido em novas estratégias de controle de versões de forma a permitir um controle de versões mais **refinado**, utilizando informações semânticas ou da estrutura dos documentos para fazer as atividades de controle das versões. Com esta abordagem, é possível disponibilizar informações mais detalhadas aos envolvidos no processo de produção de software, possibilitando inclusive a adoção de práticas mais eficazes de GCS [Lin & Reiss, 1996].

Esta abordagem tem por objetivo fornecer melhor controle sobre os documentos ou arquivos de código-fonte que estão no repositório de controle de versões, permitindo um mapeamento mais adequado da estrutura dos arquivos de computador no sistema de controle de versões. O problema existente foi bem expressado no trabalho de Nguyen, Munson e Thao [Nguyen et al., 2004b]:

“A estrutura interna de artefatos de software, especialmente códigos-fonte de programas, são muito importantes para os engenheiros de software desenvolverem e manterem softwares de alta qualidade. Entretanto, os sistemas de controle de versões e gerenciamento de configurações existentes geralmente tratam um sistema

de software como um conjunto de arquivos em diretórios sobre um sistema de arquivos. Eles geralmente desconsideram a estrutura lógica de documentação e código-fonte de programas tratando-os como um conjunto de linhas para controle de versões. Além disso, é criado um empecilho para os desenvolvedores, pois o domínio de implementação (nível lógico) e o domínio de controle de versões (nível de arquivo) requerem modelos mentais diferentes.”

Abordagens para aplicar controle de versões refinado têm sido alvo de investigações desde o final da década de 1980. Em 1988, foi proposto o sistema Orwell, cujo objetivo era oferecer suporte para o controle de versões de sistemas desenvolvidos em *Smalltalk*, permitindo o controle de versões de classes e métodos [Thomas & Johnson, 1988]. Em 1993, já havia sido identificado que os sistemas tradicionais “trabalham em arquivos completos que geralmente são unidades muito maiores do que é afetado por uma mudança única” e foi proposto um sistema para resolver o problema [Magnusson et al., 1993], que posteriormente foi substituído pelo COOP/Orm [Magnusson & Asklund, 1996].

Além do Orwell e do COOP/Orm, outros trabalhos foram propostos com o objetivo de permitir o controle de versões com granulosidade fina. Neste capítulo é apresentada uma análise comparativa dos trabalhos. Os sistemas que foram estudados foram o Molhado [Nguyen et al., 2004c], POEM [Lin & Reiss, 1996], CoEd [Bendix et al., 1998], Coop/ORM [Asklund, 2002] e Stellation [Chu-Carroll et al., 2002].

A partir do momento que as ferramentas de controle de versões refinado analisam a estrutura dos arquivos que estarão sob controle de versões, pode ser implementado o rastreamento de dependência entre arquivos de um repositório, desde que essa dependência seja declarada, de alguma forma, na gramática do arquivo. Por exemplo, no caso da linguagem de programação Java a palavra reservada “*import*” é utilizada para dizer que a classe sendo implementada depende de outra classe, que pode também estar no repositório. As ferramentas podem oferecer métodos mais ou menos flexível para definição destas regras de análise e configuração de refinamento – tal característica pode ser considerada a “flexibilidade de definição da gramática”.

Além disso, em qualquer ferramenta de suporte ao desenvolvimento de software deve-se levar em consideração o impacto da implantação de tal ferramenta num ambiente real de desenvolvimento de

software. Este impacto deve ser medido de diversas formas, e um dos fatores que influenciam é a falta de adaptação das ferramentas a serem implantadas ao processo e ferramentas já utilizados no ambiente de desenvolvimento de software. Esta característica pode ser considerada a “flexibilidade de uso” da ferramenta.

O sistema Molhado [Nguyen et al., 2004a] é apresentado como um sistema para GCS e controle de versões de **hiperdocumentos**. O foco do sistema é de fornecer informações detalhadas de hiperdocumentos. Este sistema foi projetado utilizando um outro projeto, chamado Fluid, que implementa mecanismos genéricos para controle de versões de informações em nós genéricos, utilizando um padrão que foi batizado de *node-slot pattern*<sup>2</sup>, além de fornecer meios para realizar o armazenamento destes nós e seu controle de versões. Portanto, a camada de controle de versões e armazenamento das informações no Molhado é feita utilizando o Fluid. Esta camada implementa mecanismos de armazenamento de dados primitivos, além de seu respectivo controle de versões. Como é descrito pelos autores do trabalho [Nguyen et al., 2004a], “(Fluid) oferece uma combinação de um modelo de versões e um modelo de dados primitivos para permitir o controle de versões de muitos tipos de dados”. Este sistema possui baixa flexibilidade de uso e também de definição da gramática, por não permitir a criação de regras para gramáticas diferenciadas nem permitir a extensão da ferramenta.

O sistema POEM [Lin & Reiss, 1996] – *Programmable Object-centered EnvironMent* – é constituído por um ambiente de desenvolvimento capaz de controlar as versões de itens menores do que arquivos, como classes e métodos. Ele é um ambiente de programação voltado para o desenvolvimento de programas na linguagem C++, e o mapeamento entre a gramática do documento e seu modelo de versões é programado internamente no sistema, não sendo possível redefinir estas regras. Quanto à possibilidade de estender, o POEM foi projetado como um ambiente de programação completo, e não disponibiliza qualquer API para que possa ser integrado em ferramentas já utilizadas em ambientes de desenvolvimento de software.

O CoEd é apresentado pelos seus autores como um sistema de controle de versões refinado para documentos hierárquicos [Bendix et al., 1998]. No entanto, analisando as características da ferramenta, observou-se que ela é uma ferramenta focada um

tipo específico de documentos hierárquicos, que são os textos em Latex. O sistema CoEd funciona de forma completamente dependente de sua interface gráfica, não tendo sido projetado para ser integrado em soluções que os usuários já podem estar acostumados a utilizar para escrever seus textos em Latex. Outro ponto negativo do projeto é que ele não suporta a linguagem Latex como um todo. Por exemplo, não é possível utilizar os comandos `\include` do Latex, i.e., os textos devem ser escritos utilizando sempre apenas um arquivo único.

COOP/Orm [Magnusson & Asklund, 1996] é um sistema que foi inspirado no projeto Mjolner [Knudsen et al., 1994], um sistema para desenvolvimento de software orientado a objetos. Em 1993, Magnusson et al. [Magnusson et al., 1993], cujos trabalhos são focados em desenvolvimento colaborativo de software e outras atividades relacionadas à co-operação, identificaram que “o controle de versões refinado é uma técnica crucial para o suporte da evolução de software por permitir o trabalho de times geograficamente distribuídos de engenheiros de software” [Magnusson et al., 1993]. Apesar de ter sido identificada esta melhoria em relação ao controle de versões refinado para projetos em que os desenvolvedores estão geograficamente distribuídos, Reis observou, dez anos depois, que os projetos de software livre – maiores exemplos dos projetos com desenvolvedores distribuídos – não utilizavam técnicas de controle de versões refinado, e, muitas vezes, nem mesmo realizavam qualquer atividade de controle de versões [Reis, 2003]. Após terem realizado estas investigações iniciais, os pesquisadores envolvidos no grupo de Magnusson criaram o sistema COOP/Orm, cujo objetivo é auxiliar o desenvolvimento colaborativo de software, através da implementação de técnicas que permitem o controle de versões refinado, além de diversas outras características voltadas para desenvolvimento colaborativo – como um modelo de ciência do projeto, segundo o qual cada desenvolvedor envolvido pode estar sempre ciente do que está ocorrendo no projeto como um todo.

A descrição de uma versão atualizada do COOP/Orm foi utilizada para análise [Asklund, 2002]. Apesar do foco do sistema ser a colaboração entre desenvolvedores, o conceito de controle de versões refinado é bastante explorado e é a partir dele que são oferecidas outras ferramentas que têm o objetivo de melhorar as experiências de colaboração. Para fornecer o controle de versões refinado, foi claramente definido um modelo para o

<sup>2</sup>Foi mantido o termo original que denomina o padrão. A referência bibliográfica utilizada no artigo que cita o padrão era do tipo “em preparação”, e não foram encontradas maiores informações sobre este padrão. Na página citada, do projeto Fluid – <http://fluid.cs.cmu.edu> –, também não foram encontradas referência ao padrão *node-slot*

controle de versões refinado, chamado UEVM. Apesar deste modelo ter sido apresentado anteriormente [Asklund et al., 1999], Asklund o apresentou com correções e atualizações, principalmente no que diz respeito à propagação de versões e preocupações adicionais com as características que permitem ao modelo de versões não sofrer a explosão combinatória [Asklund, 2002]. Em relação ao mapeamento da gramática de documento para o modelo de versões, a ferramenta COOP/Orm suporta este mapeamento da seguinte forma: deve-se utilizar a notação EBNF para representar a gramática da linguagem utilizada pelo documento, conforme o nível de granulosidade desejado e, então, deve-se utilizar outro arquivo na notação EBNF para realizar o mapeamento entre os nós representados no arquivo da gramática da linguagem e os nós do modelo de documentos do UEVM. Este processo não foi automatizado por nenhuma ferramenta, de forma que os usuários do COOP/Orm precisam conhecer o modelo de documentos.

Quanto às facilidades de extensão e adaptação do sistema, o COOP/Orm não disponibiliza nenhuma forma para estender a ferramenta. Não são fornecidas APIs nem é possível executar as operações via linha de comando, sendo que a utilização do modelo implementado pelo COOP/Orm é dependente da ferramenta, não sendo possível realizar sua integração com outras IDEs ou ferramentas externas – por exemplo, ferramenta de auditoria de versões externa. Devido ao fato do sistema ter como foco principal as atividades de colaboração que não foram abordadas nesta análise, ela é bastante complexa e possui diversas outras funcionalidades dependentes da atividade de controle de versões refinado, e torna-se compreensível a não implementação de APIs e outras formas de estender o sistema.

Por fim, o sistema Stellation [Chu-Carroll et al., 2002], originalmente chamado Coven [Chu-Carroll & Sprenkle, 2000], é um sistema que começou a ser desenvolvido em laboratórios de pesquisa da IBM com o objetivo de oferecer controle de versões refinado para arquivos de computador, principalmente código-fonte de programas.

O sistema analisa a estrutura lógica dos arquivos de computador e armazena itens lógicos – controle de versões mais refinado que arquivos. O menor nível documentado é o de métodos e atributos das classes. Stellation foi disponibilizado como um plugin para a IDE **Eclipse** em 2002, mas atualmente não encontra-se mais disponível. As informações sobre o sistema não têm sido atualizadas desde 2004.

Um dos conceitos utilizados pelo sistema é o de arquivos de código-fonte virtuais, que é o mapea-

mento feito dos arquivos no sistema. Este conceito assemelha-se à organização dos arquivos no sistema de arquivos, mas diferencia-se pelo fato de poder mapear unidades lógicas dentro dos arquivos – por exemplo, os métodos e atributos das classes. Este sistema possui baixa flexibilidade de definição de gramática, e, em relação à facilidade de utilização, disponibiliza comandos via linha de comando, que podem ser eventualmente utilizados para realizar a integração.

Na próxima seção, será apresentada a abordagem deste trabalho, que envolve um modelo para realização de controle de versões refinado e flexível, além de um sistema, chamado *Phoca*, que foi implementado para validar este modelo.

## 4 Phoca

*Phoca* é um sistema que foi desenvolvido com o objetivo de fornecer suporte a controle de versões refinado e flexível. Para obter este resultado, foi necessário criar um modelo que suportasse os requisitos de refinamento e flexibilidade que eram almejados, o que envolvia a flexibilidade de definição de gramática e também a flexibilidade de uso da ferramenta.

Foi desenhado o modelo conceitual da ferramenta, inspirado no UEVM – *Unified Extensional Versioning Model*. Este modelo consiste de três partes principais: um modelo de documentos, modelo de versões e meta-modelo de documentos.

O modelo de documentos possui nós de documento, composição, conteúdo e ligação. Estes nós são utilizados da seguinte forma: arquivos de computador são mapeados para nós de documento; partes estruturais do arquivo são mapeadas para nós de composição e o conteúdo todo é mapeado para nós de conteúdo. O meta-modelo de documentos possui as informações que definem as regras segundo as quais um arquivo de computador será mapeado para este modelo de documento. Na Figura 4 é apresentado o diagrama de classes do meta-modelo de documentos proposto.

O modelo de versões é idêntico ao que foi proposto no UEVM. Ele utiliza o conceito de propagação de mudanças, segundo o qual as mudanças realizadas num nó são propagadas para o nó pai, com preservação de mudanças em sessões. Portanto, no caso de alteração de qualquer parte de conteúdo do arquivo, o nó referente ao documento sempre possui uma versão criada – a não ser que uma nova versão já tenha sido criada na mesma sessão.

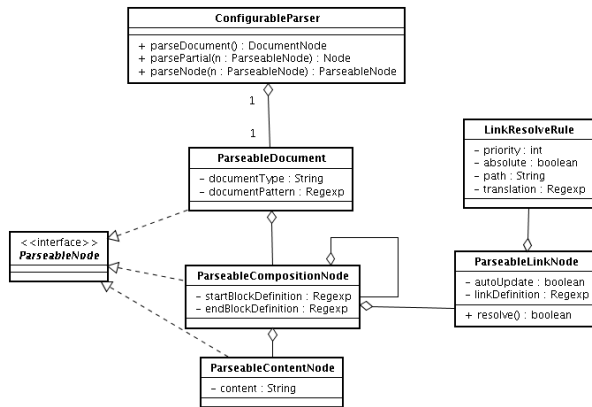


FIGURA 7: Meta-modelo de documentos

Para validar este modelo – composto pelos modelos apresentados – foi implementado o sistema *Phoca*. A arquitetura deste sistema é apresentada na Figura 4. Na figura foram representadas duas instâncias da parte cliente da aplicação para ilustrar a funcionalidade de sincronização entre clientes.

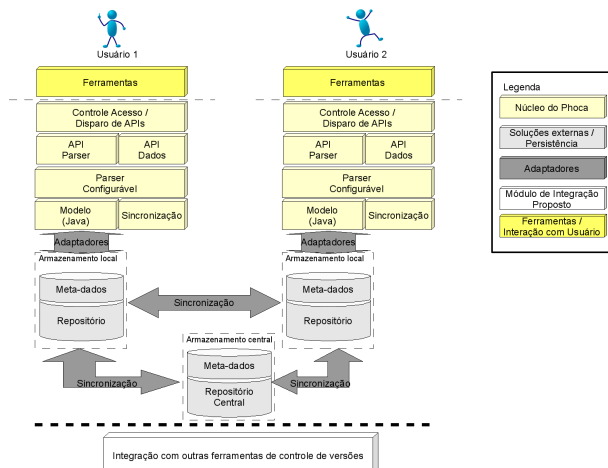


FIGURA 8: Arquitetura do sistema *Phoca*

Observa-se na Figura 4 que o sistema é composto por diversas partes, e essas partes são compostas por módulos. O núcleo refere-se à parte principal do sistema, no qual todas as funcionalidades estão implementadas. A camada de ferramentas é independente e refere-se às ferramentas de interação com o usuário, as soluções externas são ferramentas de terceiros de banco de dados e controle de versões, os adaptadores são utilizados para permitir a independência de soluções externas, e, por fim, o módulo de integração que foi proposto proporciona a integração com ferramentas de controle de versões tradicionais.

O núcleo é composto pelos seguintes módulos:

- **Modelo:** *JavaBeans* que representam as informações de modelo, como o meta-modelo de documentos, modelo de documentos e modelo de versões. O modelo de versões, como apresentado, é composto por um conjunto de regras; ele foi implementado de forma integrada ao modelo de documentos, utilizando-se o padrão de projetos *Observer* [Gamma et al., 1995].
- **Sincronização:** implementa funcionalidades de sincronização entre repositórios, incluindo sincronização entre repositórios de usuários sem intervenção do repositório central
- **Parser configurável:** é o módulo que implementa o parser do sistema, que foi feito de forma a possibilitar a flexibilidade de definição de linguagem. Na Figura 4 é ilustrado o funcionamento do parser configurável, que possui como entrada o meta-modelo de documentos e o arquivo a ser analisado e, como saída, o arquivo mapeado para o modelo de documentos. Deve-se ressaltar que a operação inversa – obtenção do arquivo a partir do modelo de documentos, ou de parte do arquivo – não envolve qualquer participação do parser configurável. Destaca-se ainda o fato de que o parser faz o mapeamento dos nós de ligação entre arquivos, e, para isso, no meta-modelo de documentos existe uma classe para armazenar as regras que devem ser utilizadas para resolução de links. Além disso, a atualização de links pode ser feita de forma automática ou manual quando o arquivo de destino do link é atualizado.
- **API Parser:** implementa o padrão de projetos *Facade* e possui operações de utilização e configuração do parser.
- **API Dados:** implementa o padrão de projetos *Facade* e possui operações de manipulação dos arquivos. É esta API também que possui acesso a operações de sincronização. Ela utiliza o módulo de modelo e realiza a maioria de suas operações sobre este módulo.
- **Dispatcher de APIs e controle de acesso:** o banco de meta-dados possui informações de permissões dos usuários sobre permissões de utilização das funcionalidades do *Phoca*. Este módulo permite que seja verificado este acesso quando são invocadas funcionalidades das APIs envolvidas. Para isso, ele foi implementado utilizando o padrão de projeto *Command* – assim, todas as funcionalidades podem



ser registradas no banco de dados, e o controle de acesso pode ser realizado. Desta forma, o dispatcher fica independente das funcionalidades e, mesmo que novas funcionalidades sejam adicionadas às APIs, este módulo permite que se mantenha a compatibilidade, além de permitir que as operações de controle de acesso sejam feitas de forma bastante simples, apenas com registros do banco de dados.

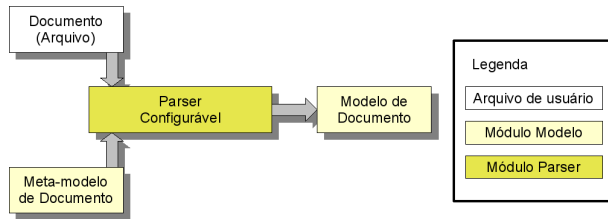


FIGURA 9: Ilustração de funcionamento do parser configurável

Vale ainda ressaltar que a forma de funcionamento do parser é com análise em passos, que envolve uma análise no formato top-down dos arquivos submetidos para análise. Na Figura 4 é exibido um exemplo de análise de um documento Latex.

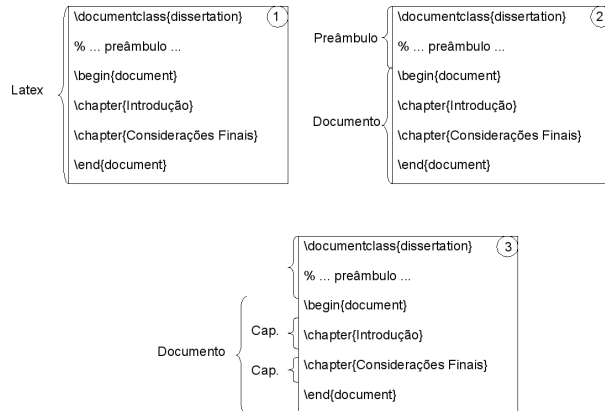


FIGURA 10: Exemplo de parser aplicado em documento Latex

Em relação às ferramentas externas, existe um banco de dados que armazena meta-dados e o repositório de controle de versões. Na implementação de referência, foi utilizado o banco de dados HSQLDB<sup>3</sup> e o repositório de controle de versões Mercurial [Mercurial, 2007].

Para realizar a troca das ferramentas externas, basta que sejam também implantados adaptadores

para estas novas ferramentas. O conceito de adaptadores tem se popularizado, e é utilizado, por exemplo, no Hibernate. O *Phoca* é entregue juntamente com interfaces dos adaptadores. Portanto, para se criar um novo adaptador para as ferramentas externas basta que as interfaces sejam implementadas, mapeando, portanto, métodos utilizados pelo núcleo do sistema em funcionalidades das ferramentas. As funcionalidades utilizadas são funcionalidades comuns a sistemas de bancos de dados e de controle de versões.

Em relação às ferramentas, o sistema é entregue junto com algumas ferramentas que possibilitam sua utilização, sendo a ferramenta de linha de comando **phc** e as ferramentas gráficas para edição do meta-modelo e para edição de documentos. A ferramenta de edição de documentos foi utilizada para validar a ferramenta implementada, embora seja extremamente simples e não deva ser utilizada em ambientes de produção. A ferramenta de edição do meta-modelo deve ser utilizada para configurar a gramática das linguagens sendo utilizada. Na Figura 4 são exibidas as ferramentas de edição de meta-modelo e edição de documentos.

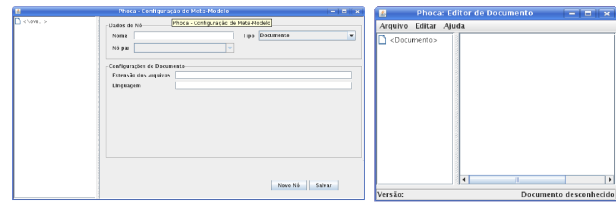


FIGURA 11: Configurador de meta-modelos e editor de documentos

A ferramenta **phc**, a ser utilizada via linha de comando, além de validar a API, permite que seja feita utilização do sistema via linha de comando. Os principais comandos disponibilizados para utilização são:

- checkout do repositório central: pode ser realizado o checkout de forma refinada. Portanto, pode-se efetuar o checkout apenas de algum nó específico e seus filhos, ou apenas da estrutura de um nó ou conjunto de nós – ou mesmo do repositório
- operações sobre o meta-modelo: operações de atualização de meta-modelo, troca de meta-modelo para determinado arquivo, checkout de meta-modelo

<sup>3</sup>Página oficial do projeto: <http://www.hsqldb.org>

- operações sobre nós de ligação: atualização de nós de ligação, configuração da atualização (automática ou manual)
- sincronização: sincronização entre usuários

Para validar a ferramenta, foram feitos alguns testes iniciais, como estudos de caso. Foi feito estudo de caso para linguagem Java, além de uma proposta que valida a flexibilidade da ferramenta.

#### 4.1 Estudos de caso

Para realização dos testes e validação inicial do presente trabalho, foram feitos alguns estudos de caso.

Um destes estudos envolveu a configuração da linguagem de programação Java, com detalhamento de classes e métodos, além do acompanhamento de links. Na Figura 4.1 é exibido o editor de documentos com a classe Java já mapeada e sendo editada.

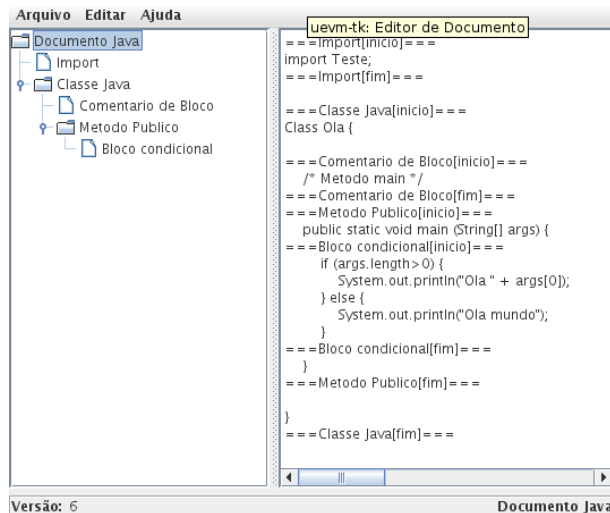


FIGURA 12: Tela do editor de documento com uma classe Java carregada

Para mostrar a flexibilidade do sistema, foi feita uma proposta de utilização do *Phoca* para controle de versões refinado das informações coletadas por sistemas cientes de contexto. Para isso, foi planejada uma nova ferramenta – configurador automático de meta-modelo – que, a partir de arquivos XML Schema ou estrutura de classes Java, configuraria o meta-modelo e possibilitaria a captura de informações de contexto além de informações relacionadas à configuração do contexto em que as informações foram capturadas. Este tipo de tarefa pode ser feito também com a utilização de sistemas de banco de dados, mas de forma menos completa e mais trabalhosa. Na Figura 4.1 é apresentada uma esquematização desta proposta.

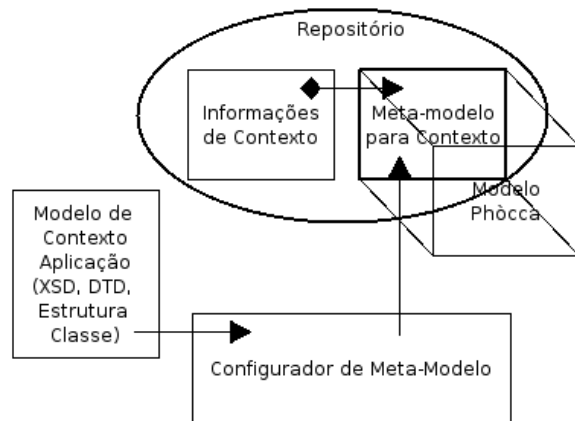


FIGURA 13: Implementação genérica proposta - controle de versões para aplicações cientes de contexto

## 5 Conclusões

Neste trabalho, foi apresentado um modelo e um sistema para realização de controle de versões refinado e flexível. Foram inicialmente identificadas as vantagens que podem ser oferecidas por sistemas deste tipo, e, então, foi feito o modelo que permitisse a realização deste tipo de tarefa.

As principais contribuições do trabalho constituem na criação de um modelo para realização do controle de versões e implementação da ferramenta de código-aberto que viabiliza este modelo, além do foco que foi dado em flexibilidade tanto da ferramenta como da definição da gramática dos documentos que podem ser criados. Como trabalhos futuros a este trabalho, foram identificadas algumas possibilidades, como:

- Realização de experimentos com usuários, para validar a usabilidade do projeto e ter feedback para melhorá-lo cada vez mais
- Criação de assistente para criação automática de meta-modelos
- Criação de ferramenta para relatórios gerenciais, que possibilite mostrar as principais funcionalidades e vantagens de se ter dados de controle de versões refinado

**Agradecimentos.** À CAPES, pelo apoio financeiro.

## Referências

- [Ambriola et al., 1990] Ambriola, V., Bendix, L., & Ciancarini, P. (1990). The evolution of configuration management and version control. *Software Engineering Journal*, 5(6):303–310.
- [Askund, 2002] Askund, U. (2002). *Configuration Management for Distributed Development in an Integrated Environment*. PhD thesis, Lund University, Lund, Suécia.
- [Askund et al., 1999] Askund, U., Bendix, L., Christensen, H. B., & Magnusson, B. (1999). The unified extensional versioning model. In *SCM-9: Proceedings of the 9th International Symposium on System Configuration Management*, pages 100–122, London, UK. Springer-Verlag.
- [Bendix et al., 1998] Bendix, L., Larsen, P. N., Nielsen, A. I., & Petersen, J. L. S. (1998). Coed - a tool for versioning of hierarchical documents. In *ECOOP '98: Proceedings of the SCM-8 Symposium on System Configuration Management*, pages 174–187, London, UK. Springer-Verlag.
- [Berlack, 1991] Berlack, H. R. (1991). *Software Configuration Management*. Wiley series in Software Engineering Practice. Wiley, Estados Unidos, 2 edition.
- [Blackwell et al., 2006] Blackwell, J. et al. (2006). *Introduction to Bzr*. Disponível em <http://bazaar-vcs.org/IntroductionToBzr>.
- [Cederqvist et al., 2007] Cederqvist et al. (2007). *Version Managment with CVS*. EUA. Disponível online em: <http://ximbiot.com/cvs/manual/>.
- [Chu-Carroll & Sprenkle, 2000] Chu-Carroll, M. C. & Sprenkle, S. (2000). Coven: brewing better collaboration through software configuration management. *SIGSOFT Softw. Eng. Notes*, 25(6):88–97.
- [Chu-Carroll et al., 2002] Chu-Carroll, M. C., Wright, J., & Shields, D. (2002). Supporting aggregation in fine grained software configuration management. *SIGSOFT Softw. Eng. Notes*, 27(6):99–108.
- [Conradi & Westfechtel, 1998] Conradi, R. & Westfechtel, B. (1998). Version models for software configuration management. *ACM Comput. Surv.*, 30(2):232–282.
- [Estublier et al., 2005] Estublier, J., Leblang, D., van der Hoek, A., Conradi, R., Clemm, G., Tichy, W., & Wiborg-Weber, D. (2005). Impact of software engineering research on the practice of software configuration management. *ACM Trans. Softw. Eng. Methodol.*, 14(4):383–430.
- [Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns*. Addison-Wesley Professional.
- [Garzik, 2005] Garzik, J. (2005). *Kernel Hackers' Guide to git*. Disponível em <http://linux.yyz.us/git-howto.html>.
- [Glasser, 1978] Glasser, A. L. (1978). The evolution of a source code control system. In *Proceedings of the software quality assurance workshop on Functional and performance issues*, pages 122–125.
- [IEEE Std 610, 1991] IEEE Std 610 (1991). *IEEE standard computer dictionary, A compilation of IEEE standard computer glossaries*. IEEE, New York.
- [Keyes, 2004] Keyes, J. (2004). *Software Configuration Management*. Auerbach, New York, NY, USA, 1 edition.
- [Knudsen et al., 1994] Knudsen, J. L., Magnusson, B., Lofgren, M., & Madsen, O. L. (1994). *Object Oriented Software Development Environments: The Mjolner Approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
- [Kroah-Hartman, 2002] Kroah-Hartman, G. (2002). Kernel korner: the kernel hacker's guide to source code control. *Linux J.*, 2002(101):11.
- [Lin & Reiss, 1996] Lin, Y.-J. & Reiss, S. P. (1996). Configuration management with logical structures. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 298–307, Washington, DC, USA. IEEE Computer Society.
- [Lindsay et al., 1997] Lindsay, P., Liu, Y., & Owen-Traynor (1997). A generic model for fine grained configuration management including version control and traceability. In *Software Engineering Conference, 1997. Proceedings. 1997 Australian*, Sydney, NSW. IEEE.
- [Magnusson & Askund, 1996] Magnusson, B. & Askund, U. (1996). Fine grained version control of configurations in coop/orm. In *ICSE '96: Proceedings of the SCM-6 Workshop on System Configuration Management*, pages 31–48, London, UK. Springer-Verlag.

- [Magnusson et al., 1993] Magnusson, B., Asklund, U., & Minör, S. (1993). Fine-grained revision control for collaborative software development. In *SIGSOFT '93: Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering*, pages 33–41, New York, NY, USA. ACM.
- [Mercurial, 2007] Mercurial (2007). Site oficial do projeto mercurial. Disponível em <http://www.selenic.com/mercurial/wiki/>.
- [Nguyen et al., 2004a] Nguyen, T. N., Munson, E. V., & Boyland, J. T. (2004a). The molhado hypertext versioning system. In *HYPERTEXT '04: Proceedings of the fifteenth ACM conference on Hypertext and hypermedia*, pages 185–194, New York, NY, USA. ACM.
- [Nguyen et al., 2004b] Nguyen, T. N., Munson, E. V., Boyland, J. T., & Thao, C. (2004b). Flexible fine-grained version control for software documents. In *Proceedings of Software Engineering Conference, 2004. 11th Asia-Pacific*. IEEE.
- [Nguyen et al., 2004c] Nguyen, T. N., Munson, E. V., & Thao, C. (2004c). Fine-grained, structures configuration management for web projects. In *Proceedings of WWW2004*, pages 433–442, New York, NY. ACM Press.
- [RCS, 2007] RCS (2007). Gnu revision control system. Site oficial do projeto: <http://www.gnu.org/software/rcs/>. Visitado em outubro/2007.
- [Reis, 2003] Reis, C. R. (2003). Caracterização de um processo de software para projetos de software livre. Master's thesis, ICMC, USP, São Carlos, Brasil.
- [Rochkind, 1975] Rochkind, M. J. (1975). The source code control system. *IEEE Trans. Software Eng.*, 1(4):364–370.
- [Thomas & Johnson, 1988] Thomas, D. & Johnson, K. (1988). Orwell - a configuration management system for team programming. In *OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 135–141, New York, NY, USA. ACM.
- [Tichy, 1982] Tichy, W. F. (1982). Design, implementation, and evaluation of a revision control system. In *ICSE '82: Proceedings of the 6th international conference on Software engineering*, pages 58–67, Los Alamitos, CA, USA. IEEE Computer Society Press.